CS3025 Compiladores

Semana 11:
Generacion de Codigo:
The Basics
24 Octubre 2023

Igor Siveroni

Objetivos

Hemos trabajado con los siguientes lenguajes:

- IMP: Lenguaje imperativo.
- SVML: Lenguaje para una maquina de pila virtual (SVM)

Llamemos a IMP y SVML nuestros lenguajes fuente y objeto, respectivamente.
En lo queda del curso,

- Describiremos como transformar un programa escrito en IMP a un programa equivalente escrito en SVM.
- A este proceso lo llamamos Generación de Código Objeto.

Para empezar, solo trabajaremos con una versión simplificada de IMP – IMP0 – una version que tiene declaración de variables mas no declaración de funciones.

En esta clase, sentaremos las bases de: Generación de código objeto para IMP0

Generacion de Codigo Objeto

Dado un lenguaje de programación L1 definido por una sintaxis y semántica, hasta el momento podemos:

- Escribir un analizador léxico y sintáctico para verificar si el programa satisface la gramática.
- Generar un árbol sintáctico abstracto como representación intermedia.
- Realizar el análisis semántico del programa original (type-checking)
- Interpretar el programa.

Dado otro lenguaje L2 , de preferencia de “menor nivel”, definido por otra sintaxis y gramática, y con los mismos componentes descritos arriba,
¿Que nos interesa hacer?

- Tomar un programa escrito en L1 y generar un programa “equivalente” en el lenguaje L2.
- “Equivalente” quiere decir que “ejecutan a lo mismo”.

A este proceso lo llamamos “**Generacion de Codigo Objeto**”. En las siguientes clases estudiaremos técnicas para poder realizar esta operación.

El lenguaje objeto: SVML, lenguaje para una maquina de pila virtual

Empecemos definiendo nuestro lenguaje objeto, SVML.
SVML, el lenguaje de la maquina de pila virtual SML, esta definido por la siguiente sintaxis:

```
PSVM ::= Instruction+
Instruction ::= Label? (<instr0> | <instr1>  <num> | <instr2> <id>) <eol>
<instr0> ::= skip | pop | dup | swap | add | sub | mul | div
             gt | ge | lt | le | eq | print
<instr1> ::= push | store | load
<instr2> ::= jmpz | jmpn| goto
```

Un programa SVML esta compuesto por una secuencia de instrucciones separadas por cambio de línea.
La sintaxis de arriba divide a las instrucciones en 3 tipos, dependiendo del numero (0 o 1) de argumentos y tipo de argumento (num o id) que aceptan.
Ademas, cada instrucción puede estar anotada por un Label (<id>).

Este es un subconjunto de SVML – será suficiente por ahora.

SVML: Ejemplo

```
push 7
store 5
push 0
LENTRY: load 5
push 0
le
jmpn LEND
load 5
add
load 5
push 1
sub
store 5
goto LENTRY
LEND: skip
```

SVML: Semantica – La pila de datos

La característica principal de la SVM es el uso de una pila de operandos: todas las operaciones de la SVM toman sus argumentos de la pila de datos S (stack).

Para mas claridad, será conveniente expresar la pila como `S = d1:…:dn`, donde `dn` corresponde al ultimo elemento de la pila.
También será útil escribir `S:top` y `S:next:top` donde, en ambos casos, `top` es el ultimo elemento de la pila.
Las operaciones que manipulan directamente la pila son:

- `push n`: coloca al numero n arriba de la pila.
  `S` cambia a `S:n`
- `pop`: remueve el ultimo elemento de la pila.
  `S:top` cambia a `S`
- `dup`: duplica el ultimo elemento de la pila.
  `S:n` cambia a `S:n:n`
- `swap`: intercambia los 2 últimos elementos de la pila.
  `S:next:top` cambia a `S:top:next`

SVML: Semantica – Operaciones binarias

Las operaciones aritméticas y de comparación toman, y remueven, 2 argumentos de la pila, y dejan el resultado encima de la pila. Dada la operación `op`, podemos expresar la ejecución de la operación de la siguiente manera:

`S:next:top` cambia a `S(next |op| top)`

Donde op = `{ add, sub, mul, div, lt, le, gt, ge, eq }`, y

- |op| representa la operación aritmética o comparación eg. + o <.
- Las comparaciones evalúan a 0 o 1.
- Es importante notar el orden de los operandos. Por ejemplo
  `push 4; push 2; sub;` deja `(4-2)` arriba de la pila, y
  `push 5; push 10; lt;` deja `(5<10)` arriba de la pila, es decir, 0.

SVML: Semantica – Cambios en el flujo de control

La ejecución de un programa en SVM, por defecto, es secuencial. Si consideramos a un programa de la SVM como una secuencia de instrucciones P indexadas por el registro PC (program counter), entonces el loop de ejecución de la maquina virtual puede escribirse así:

```
while(PC < |P|) { I=P[PC]; PC++; execute(I); }
```

Existen 3 instrucciones que modifican el flujo normal del programa.

- `goto L`: Cambia el flujo del programa a la instrucción anotada con (el label) `L`.
- `jmpz L` : Cambia el flujo del programa a la instrucción anotada con (el label) `L`, si el elemento `top` de la pila es 0. Remueve el ultimo elemento de la pila.
- `jmpn L` : Cambia el flujo del programa a la instrucción anotada con (el label) `L`, si el elemento `top` de la pila NO es 0. Remueve el ultimo elemento de la pila.

En los 3 casos, el valor a PC cambia a `idx(L)`, la posición de la instrucción anotada por `L`.
A `jmpz` y `jmpn` los llamamos saltos condicionales.

SVML: Semantica – Acceso a Memoria

Además de la pila de datos, la SVM posea una sección de memoria que puede ser leída y escrita por media de un índice: la dirección de memoria. Representemos, por ahora, a la memoria como un arreglo `M`. Dada una dirección de memoria `a`, podemos acceder al valor guardado en la dirección `a` escribiendo `M[a]`. Del mismo modo podemos guardar un valor en la posición `a` escribiendo `M[a] := v`.

Las instrucciones que acceden ala memoria M de la SVM son:

- `load a`: Lee el valor guardado en memoria en la dirección a y lo coloca encima de la pila.

  `S` cambia a `S:M[a]`

- `store a` : Escribe en memoria, en la dirección a, el valor top de la pila. Remueve etop de la pila.

  `S:top` cambia a `S` `M[a] = top`

IMP0, nuestro lenguaje fuente

Hemos estado trabajado con el lenguaje IMP, un lenguaje con declaraciones de funciones globales.
Para esta primera parte de la fase de generación de código, trabajaremos con una versión simplificada de IMP, IMP0:

```
Program ::= Body
Body ::= VarDecList StmList
VarDecList ::= (VarDec)*
VarDec ::= “var” Type VarList “;”
Type ::= bool | int
StmList ::= Stm (“;” Stm)*
```

Donde un programa es un bloque `Body`, definido como una lista de declaraciones de variables (globales para el primer bloque) seguidas de una secuencia no-vacia de sentencias.

IMP0 será nuestro lenguaje fuente.

El lenguaje IMP: sentencias y expresiones

La gramática de sentencias es:

```
Stm ::=  AssignStm | PrintStm | IfStm | WhileStm | ReturnStm
AssignStm ::= id “=” Exp
PrintStm ::= “print” “(“ Exp “)”
IfStm ::= “if” Exp “then” Body [“else” Body] “endif”
WhileStm ::= “while” Exp “do” Body “endwhile”
```

La gramática de expresiones, con cuestiones de asociatividad y orden de precedencia resueltas, es:

```
Exp ::=  NumberExp | BoolExp | IdExp | CondExp | ParenthExp | BinExp
NumberExp ::= num     BoolExp ::= “true” | “false”     IdExp ::= id
BinExp ::= Exp op Exp, op in {plus, minus, mul, div, lt, leq, eq }
CondExp ::= “ifexp” “(“ Exp “,” Exp “,” Exp “)”
ParenthExp ::= “(“ Exp “)”
```

IMP-SVML: Generacion de codigo SVML (expresiones, sintaxis abstracta)

Para facilitar la especificación de las reglas de generación de codigo, expresaremos la sintaxis de IMP usando un tipo especial de sintaxis abstracta (abstract syntax).

```
e in Exp ::=
      | NumberExp(n), n in Int
      | BoolExp(b)  , b in { T, F }
      | IdExp(id)   , id in ID
      | CondExp(e0, e1, e2) |
      | BinExp(e1, e2, op) ,
                op in {plus, minus, mul, div, lt, leq, eq }
      | ParenthExp(e)
```

IMP: Generacion de Codigo

Para especificar el generación de código de programas IMP a SVML, definiremos la función de una funcion `codegen` de la siguiente manera:

```
codegen: Aenv x Exp -> Instruction+
codegen: Aenv x Stm -> Instruction+
codegen: Aenv x Body -> Instruction+
codegen: Aenv x Program -> Instruction+
```

Es decir, `codegen` mapea elementos de IMP a secuencias de instrucciones de SVML.
Para esta transformación es necesario tener un environment de direcciones de memoria Aenv

```
env in Aenv : id -> int
```

Que mantendrá un registro de las posiciones en memoria de todas las variables vivas, en cualquier punto durante la generación de código del programa.

Además, será conveniente tener una función getLabel()->Label que generara “fresh labels”.

IMP: Generacion de Codigo

Como es costumbre, desarrollaremos nuestra intuición del problema resolviéndolo con código, es decir, implementándolo primero (laboratorio), antes de presentarlo de manera mas formal.

Vayamos al laboratorio 11.2.